3 NEURAL NETWORK ARCHITECTURES

Hooman Yousefizadeh and Ali Zilouchian

3.1 INTRODUCTION

Interest in the study of neural networks has grown remarkably in the last two decades. This is due to the conceptual viewpoint regarding the human brain as a model of a parallel computation device, a very different one from a traditional serial computer. Neural networks are commonly classified by their network topology, node characteristics, learning, or training algorithms. On the other hand, the potential benefits of neural networks extend beyond the high computation rates provided by massive parallelism of the networks. They typically provide a greater degree of robustness or fault tolerance than Von Neumann sequential computers. Additionally, adaptation and continuous learning are integrated components of NN. These properties are very beneficial in areas where the training data sets are limited or the processes are highly nonlinear. Furthermore, designing artificial neural networks to solve problems and studying real biological networks (Chapter 4) may also change the way we think about the problems and may lead us to new insights and algorithm improvements.

The main goal of this chapter is to provide the readers with the conceptual overviews of several neural network architectures. The chapter will not delve too deeply into the theoretical considerations of any one network, but will concentrate on the mechanism of their operation. Examples are provided for each network to clarify the described algorithms and demonstrate the reliability of the network. In the following four chapters various applications pertaining to these networks will be discussed.

This chapter is organized as follows. In section 3.2, various classifications of neural networks according to their operations and/or structures are presented. Feedforward and feedback networks are discussed. Furthermore, two different methods of training, namely supervised and unsupervised learning, are described. Section 3.3 is devoted to error back propagation (BP) algorithm. Various properties of this network are also discussed in this section. Radial basis function network (RBFN) is a feedforward network with supervised learning, which is the subject of the discussion in section 3.4. Kohonen self-organizing as well as Hopfield networks are presented in sections 3.5 and 3.6, respectively. Finally section 3.7 presents the conclusions of this chapter.

VISIT…

LANZAROTE
Caliente.COM

3.2 NN CLASSIFICATIONS

3.2.1 Feedforward and Feedback Networks

In a feedforward neural network structure, the only appropriate connections are between the outputs of each layer and the inputs of the next layer. Therefore, no connections exist between the outputs of a layer and the inputs of either the same layer or previous layers. Figure 3.1 shows a two-layer feedforward network. In this topology, the inputs of each neuron are the weighted sum of the outputs from the previous layer. There are weighted connections between the outputs of each layer and the inputs of the next layer. If the weight of a branch is assigned a zero, it is equivalent to no connection between correspondence nodes. The inputs are connected to each neuron in hidden layer via their correspondence weights. Outputs of the last layer are considered the outputs of the network.

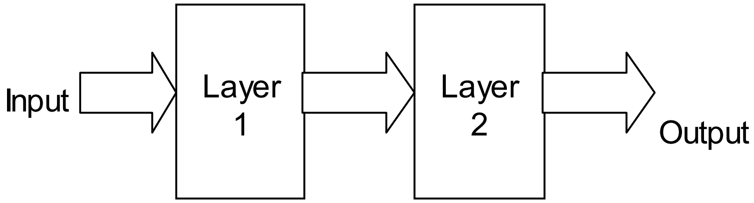


Figure 3.1: General Structure of Two-Layer Feedforward Network.

For feedback networks the inputs of each layer can be affected by the outputs from previous layers. In addition, self feedback is allowed. Figure 3.2 shows a simple single layer feedback neural network.

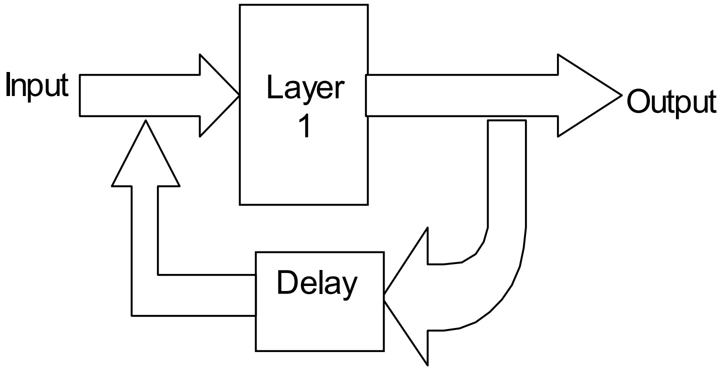


Figure 3.2: General Structure of a Sample Feedback Network.

As observed, the inputs of the network consist of both external inputs and the network output with some delays. Examples of feedback algorithms include the Hopfield network, described in detail in section 3.6, and the Boltzman Machine.

An important issue for feedback networks is the stability and convergence of the network.

3.2.2 Supervised and Unsupervised Learning Networks

There are a number of approaches for training neural networks. Most fall into one of two modes:

- *Supervised Learning*: Supervised learning requires an external teacher to control the learning and incorporates global information. The teacher may be a training set of data or an observer who grades the performance. Examples of supervised learning algorithms are the least mean square (LMS) algorithm and its generalization, known as the back propagation algorithm[1]-[4], and radial basis function network [5]-[8]. They will be described in the following sections of this chapter.

In supervised learning, the purpose of a neural network is to change its weights according to the inputs/outputs samples. After a network has established its input output mapping with a defined minimum error value, the training task has been completed. In sequel, the network can be used in recall phase in order to find the outputs for new inputs. An important factor is that the training set should be comprehensive and cover all the practical areas of applications of the network. Therefore, the proper selection of the training sets is critical to the good performance of the network.

- *Unsupervised Learning*: When there is no external teacher, the system must organize itself by internal criteria and local information designed into the network. Unsupervised learning is sometimes referred to as *self-organizing learning*, i.e., learning to classify without being taught. In this category, only the input samples are available and the network classifies the input patterns into different groups. Kohonen network is an example of unsupervised learning.

3.3 BACK PROPAGATION ALGORITHM

Back propagation algorithm is one of the most popular algorithms for training a network due to its success from both simplicity and applicability viewpoints. The algorithm consists of two phases: *Training phase and recall phase*. In the *training* phase, first, the weights of the network are randomly initialized. Then, the output of the network is calculated and compared to the desired value. In sequel, the error of the network is calculated and used to adjust the weights of the output layer. In a similar fashion, the network error is also propagated backward and used to update the weights of the previous layers. Figure 3.3 shows how the error values are generated and propagated for weights adjustments of the network.

In the *recall* phase, only the feedforward computations using assigned weights from the training phase and input patterns take place. Figure 3.4 shows both the feedforward and back propagation paths. The feedforward process is used in both recall and training phases. On the other hand, as shown in Figure 3.4(b), back propagation of error is only utilized in the training phase.

In the training phase, the weight matrix is first randomly initialized. In sequel, the output of each layer is calculated starting from the input layer and moving forward toward the output layer. Thereafter, the error at the output layer is calculated by comparison of actual output and the desired value to update the weights of the output and hidden layers.

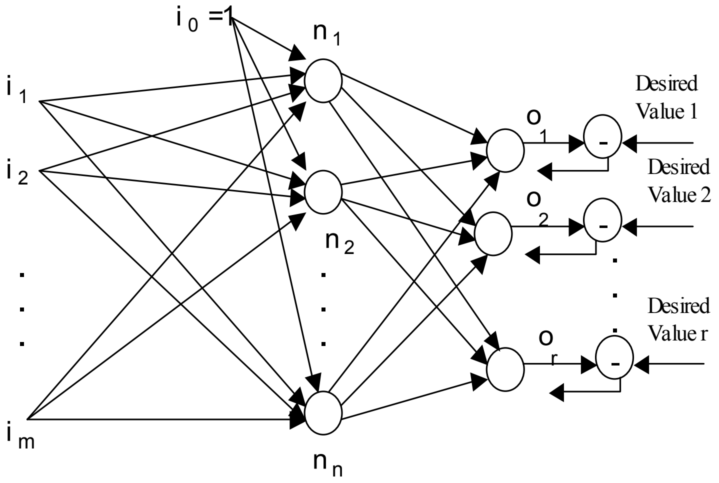
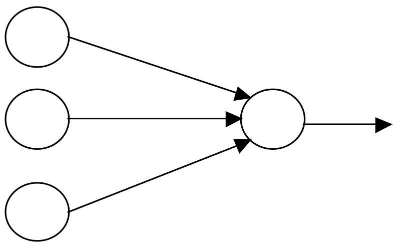
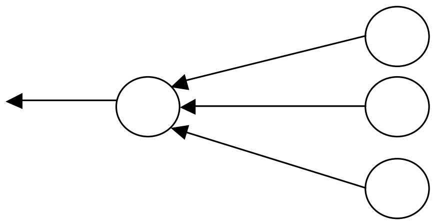


Figure 3.3. Back Propagation of the Error in a Two-Layer Network.



a) Forward propagation (Training and Recall Phase)



b) Backward propagation (Training Phase)

Figure 3.4: Forward Propagation in Recall and Training Phase and Backward Propagation in Training Phase.

There are two different methods of updating the weights. In the first method, weights are updated for each of the input patterns using an iteration method. In the second method, an overall error for all the input output patterns of training sets is calculated. In other words, either each of the input patterns or all of the patterns together can be used for updating the weights. The training phase will be terminated when the error value is less than the minimum set value provided by the designer. One of the disadvantages of back propagation algorithm is that the training phase is very time consuming.

During the recall phase, the network with the final weights resulting from the training process is employed. Therefore, for every input pattern in this phase, the output will be calculated using both linear calculation and nonlinear activation functions. The process provides a very fast performance of the network in the recall phase, which is one of its important advantages.

3.3.1 Delta Training Rule

The back propagation algorithm is the extension of the perceptron structure as discussed in the previous chapter with the use of multiple adaptive layers. The training of the network is based on the delta training rule method. Consider a single neuron in Figure 3.5.

The relations among input, activity level and output of the system can be shown as follows:

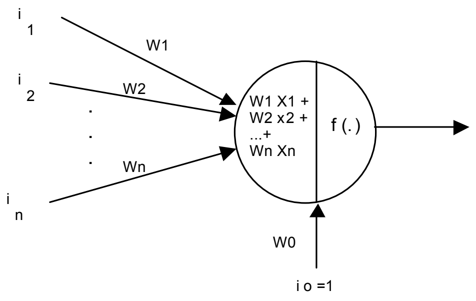


Figure 3.5: A Single Neuron.

$$a = w_0 + w_1 i_1 + w_2 i_2 + \cdots + w_n i_n \tag{3.1}$$

or in the matrix form:

$$a = w_0 + W^T I \tag{3.2}$$

$$o = f(a) \tag{3.3}$$

where W and I are weight and input vectors of the neuron, a is activity level of the neuron and o is the output of the neuron. w_0 is called bias value.

Suppose the desired value of the output is equal to d. Error e can be defined as follows:

$$e = \frac{1}{2}(d - o)^2 \tag{3.4}$$

by substituting Equations 3.2 and 3.3 into Equation 3.4, the following relation holds:

$$e = \frac{1}{2}(d - f(w_0 + W^T I))^2 \tag{3.5}$$

The error gradient vector can be calculated as follows:

$$\nabla e = -(d - o) f'(w_0 + W^T I) I \tag{3.6}$$

The components of gradient vector are equal to:

$$\frac{\partial e}{\partial w_j} = -(d - o) f'(w_0 + W^T I) I_j \tag{3.7}$$

where $f'(.)$ is derivative of activation function. To minimize the error the weight changes should be in negative gradient direction. Therefore we will have

$$\Delta W = -\eta \nabla e \tag{3.8}$$

where η is a positive constant, called learning factor. By Equations (3.6) and 3.7, the ΔW is calculated as follows:

$$\Delta W = -\eta (d - o) f'(a) I \tag{3.9}$$

For each weight j Equation 3.9 can be written as:

$$\Delta w_j = -\eta (d - o) f'(a) I_j \qquad j = 0,1,2,\ldots,n \tag{3.10}$$

Therefore we update the weights of the network as:

$$w_j(new) = w_j(old) + \Delta w_j \qquad j = 0,1,2,\ldots,n \tag{3.11}$$

For Figure 3.3, the Delta rule can be applied in a similar manner to each neuron. Through generalization of Equation 3.11 for normalized error and using Equation 3.10 for every neuron in output layer we will have:

$$w_j(new) = w_j(old) + \frac{\eta (d_j - o_j) f'(a_j) x_j}{\|X\|^2} \qquad j = 0,1,2,\ldots,n \tag{3.12}$$

where $X \in R^n$ is the input vector to the last layer, xj is the j^th element of X and ||.|| denotes L2-Norm.

The above method can be applied to the hidden layers as well. The only difference is that the o_j will be replaced by y_j in 3.12. y_j is the output of hidden layer neuron, and not the output of network.

One of the drawbacks in the back propagation learning algorithm is the long duration of the training period. In order to improve the learning speed and avoid the local minima, several different methods have been suggested by researchers. These include addition of first and second moments to the learning phase, choosing proper initial conditions, and selection of an adaptive learning rate.

To avoid the local minima, a new term can be added to Equation 3.12. In such an approach, the network memorizes its previous adjustment, and, therefore it will escape the local minima, using previous updates. The new equation can be written as follows:

$$w_{j\,(new)} = w_{j\,(old)} + \frac{\eta(d_j - o_j)f'(a_j)x_j}{\|X\|^2} + \alpha[w_{j\,(new)} - w_{j\,(old)}] \quad (3.13)$$

where α is a number between 0 and 1, namely the momentum coefficient.

Nguyen and Widrow [9] have proposed a systematic approach for the proper selection of initial conditions in order to decrease the training period of the network. Another approach to improve the convergence of the network and increase the convergence speed is the adaptive learning rate. In this method, the learning rate of the network (η) is adjusted during training. In the first step, the training coefficient is selected as a large number, so the resulting error values are large. However, the error will be decreased as the training progresses, due to the decrease in the learning rate. It is similar to coarse and fine tunings in selection of a radio station.

In addition to the above learning rate and momentum terms, there are other neural network parameters that control the network's performance and prediction capability. These parameters should be chosen very carefully if we are to develop effective neural network models. Two of these parameters are described below.

Selection of Number of Hidden Layers

The number of input and output nodes corresponds to the number of network inputs and desired outputs, respectively. The choice of the number of hidden layers and the nodes in the hidden layer(s) depends on the network application. Selection of the number of hidden layers is a critical part of designing a network and is not as straightforward as input and output layers. There is no mathematical approach to obtain the optimum number of hidden layers, since such selection is generally fall into the application oriented category. However, the number of hidden layers can be chosen based on the training of the network using various configurations, and selection of the configuration with the fewest number of layers and nodes which still yield the minimum root-mean-squares (RMS) error quickly and efficiently. In general, adding a second hidden layer improves the network's prediction capability due to the nonlinear separability property of the network. However, adding an extra hidden layer commonly yields prediction capabilities similar to those of two-hidden layer networks, but requires longer training times due to the more complex structures. Although using a single hidden layer is sufficient for solving many functional approximation problems, some problems may be easier to solve with a two-hidden-layer configuration.

Normalization of Input and Output Data Sets

Neural networks require that their input and output data be normalized to have the same order of magnitude. Normalization is very critical for some applications. If the input and the output variables are not of the same order of magnitude, some variables may appear to have more significance than they actually do. The training algorithm has to compensate for order-of-magnitude differences by adjusting the network weights, which is not very effective in many of the training algorithms such as back propagation algorithm. For example, if input variable i_1 has a value of 50,000 and input variable i_2 has a value of 5, the assigned weight for the second variable entering a node of hidden layer 1 must be much greater than that for the first in order for variable 2 to have any significance. In addition, typical transfer functions, such as a sigmoid function, or a hyperbolic tangent function, cannot distinguish between two values of x_i when both are very large, because both yield identical threshold output values of 1.0.

The input and output data can be normalized in different ways. In Chapters 7 and 15, two of these normalized methods have been selected for the appropriate applications therein.

The training phase of back propagation algorithm can be summarized in the following steps:

1. Initialize the weights of the network.
2. Scale the input/output data.
3. Select the structure of the network (such as the number of hidden layers and number of neurons for each layer).
4. Choose activation functions for the neurons. These activation functions can be uniform or they can be different for different layers.
5. Select the training pair from the training set. Apply the input vector to the network input.
6. Calculate the output of the network based on the initial weights and input set.
7. Calculate the error between network output and the desired output (the target vector from the training pair).
8. Propagate error backward and adjust the weights in such a way that minimizes the error. Start from the output layer and go backward to input layer.
9. Repeat steps 5−8 for each vector in the training set until the error for the set is lower than the required minimum error.

After enough repetitions of these steps, the error between the actual outputs and target outputs should be reduced to an acceptable value, and the network is said to be trained. At this point, the network can be used in the recall or generalization phases where the weights are not changed.

Network Testing

As we mentioned before, an important aspect of developing neural networks is determining how well the network performs once training is complete. Checking the performance of a trained network involves two main criteria: (1) how well the neural network recalls the output vector from data sets used to train the network (called the *verification step*); and (2) how well the network predicts responses from data sets that were not used in the training phase (called the *recall* or *generalization step*).

In the verification step, we evaluate the network's performance in specific initial input used in training. Thus, we introduce a previously used input pattern to the trained network. The network then attempts to predict the corresponding output. If the network has been trained sufficiently, the network output will differ only slightly from the actual output data. Note that in testing the network, the weight factors are not changed: they are frozen at their last values when training ceased.

Recall or generalization testing is conducted in the same manner as verification testing; however, now the network is given input data with which it was not trained. Generalization testing is so named because it measures how well the network can generalize what it has learned, and form rules with which to make decisions about data it has not previously seen. In the generalization step, we feed new input patterns (whose results are known to us, but not to the network) to the trained network. The network generalizes well when it sensibly interpolates these new patterns. The error between the actual and predicted outputs is larger for generalization testing and verification testing. In theory, these two errors converge upon the same point corresponding to the best set of weight factors for the network.

In the following subsection, two examples are presented to clarify various issues related to BP.

Example 3.1:

Consider the network of Figure 3.6 with the initial values as indicated. The desired values of the output are $d_0 = 0 \quad d_1 = 1$. We show two iterations of learning of the network using back propagation. Suppose the activation function of the first layer is a sigmoid and activation function of the output is a linear function.

$$f(x) = \frac{1}{1+e^{-x}} \Rightarrow f'(x) = f(x)[1 - f(x)] \tag{3.14}$$

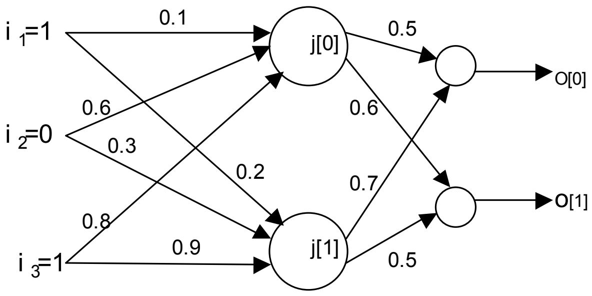


Figure 3.6: Feedforward Network of Example 3.1 with Initial Weights.

Iteration Number 1:

Step 1: Initialization: First the network is initialized with the values as shown in Figure 3.6.

Step 2: Forward calculation, using Equations (3.1–3.3):

$$\mathrm{J}[0] = f(W_j[0].I) = f(0.1*1 + 0.6*0 + 0.8*1 = f(0.9) = 0.7109$$

$$\mathrm{J}[1] = f(W_j[1].I) = f(1.1) = 0.7503$$

$$\text{O}[0] = f(W_k[0].J) = 0.5*0.7109 + 0.7*0.7503 = 0.88066$$
$$\text{O}[1] = f(W_k[1].J) = 0.6*0.7109 + 0.5*0.7503 = 0.80169$$

Step 3: According to Equation 3.5 the errors are calculated as follows:

$$\Delta k[0] = d_0 - k[0] = 0 - 0.88066 = -0.88066$$
$$\Delta k[1] = d_1 - k[1] = 1 - 0.80169 = 0.19831$$

Step 4: The updated weights of the network are calculated according to Equations 3.10 and 3.11 as follows:

$$\text{W}_{\text{k00}}(new) = \text{W}_{\text{k00}}(old) + n*\Delta k[0]*\text{f}(\text{k}[0])*j[0] =$$
$$0.5 + 1*(-0.88066) + 0.2072*0.7109 = 0.3694$$

$\text{W}_{\text{k01}}(new) = 0.56309$ $\text{W}_{\text{k10}}(new) = 0.6301$ $\text{W}_{\text{k11}}(new) = 0.5138$

$$\text{W}_{\text{j00}}(new) = \text{W}_{\text{j00}}(old) + n*\text{I}[0]*\Sigma w\Delta =$$
$$0.1 + 1*1*(0.5* -0.88066 + 0.6*0.19831) = -0.2213$$

$\text{W}_{\text{j01}}(new) = 0.6$ $\text{W}_{\text{j02}}(new) = 0.4787$ $\text{W}_{\text{j10}}(new) = -0.3173$

$\text{W}_{\text{j11}}(new) = 0.3$ $\text{W}_{\text{j12}}(new) = 0.3827$

Iteration Number 2: For this iteration the new weight values in Iteration 1 are utilized. Steps 2–4 of the previous iteration are repeated.

Step 2:

$\text{J}[0] = 0.5640$ $\text{J}[1] = 0.5163$ $\text{O}[0] = 0.4991$ $\text{O}[1] = 0.6299$

Step 3:

$\Delta k[0] = -0.4991$ $\Delta k[1] = 0.3701$

Step 4:

$\text{W}_{\text{k00}}(new) = 0.3032$ $\text{W}_{\text{k01}}(new) = 0.5025$ $\text{W}_{\text{k10}}(new) = 0.6774$

$\text{W}_{\text{k11}}(new) = 0.5751$ $\text{W}_{\text{j00}}(new) = -0.17248$ $\text{W}_{\text{j01}}(new) = 0.6$

$\text{W}_{\text{j02}}(new) = 0.5275$ $\text{W}_{\text{j10}}(new) = -0.4015$ $\text{W}_{\text{j11}}(new) = 0.3$

$\text{W}_{\text{j12}}(new) = 0.2985$

The weights after the two iterations of training of the network can be calculated as follows:

$\text{J}[0] = 0.5878$ $\text{J}[1] = 0.5257$ $\text{O}[0] = 0.4424$ $\text{O}[1] = 0.7005$

Table 3.1 summarizes the results for the training phase. As can be seen, the values of the output are closer to the desired value and the error value has been decreased. Training should be continued until the error values become less than a predetermined value as set by the designer (for example, 0.01). It should be noted that the selection of small values for maximum error level will not necessarily lead to better performance in the recall phase.

Table 3.1: Summary of Outputs and Error Norm after Iterations

Error	Initial	Iteration 1	Iteration 2
Output 1	- 0.8807	- 0.4991	- 0.4424
Output 2	0.1983	0.3701	0.2995
Error Norm	0.9027	0.6213	0.5342

Choosing a very small value for this maximum error level may force the network to learn the inputs very well, but it will not lead to better overall performance.

Example 3.1 is also solved using MATLAB as shown in Chapter 21. Below is the output result of the program.

$$\begin{matrix} Final \\ Output \end{matrix} = \begin{bmatrix} -0.0088 \\ 1.0370 \end{bmatrix} \qquad \begin{matrix} Input\ \ Layer \\ Weight \end{matrix} = \begin{bmatrix} -0.0255 & 0.6 & 0.6475 \\ 0.0763 & 0.3 & 0.7763 \end{bmatrix}$$

$$\begin{matrix} Hidden\ \ Layer \\ Weight \end{matrix} = \begin{bmatrix} 0.1170 & 0.2897 \\ 0.4987 & 0.4159 \end{bmatrix} \qquad \begin{matrix} Bias \\ Weight \end{matrix} = \begin{bmatrix} -0.1255 \\ -0.1237 \end{bmatrix}$$

As observed, only four iterations are needed to complete the training task for this example. (In this case, the training sets include only one input output set, so each epoch is equivalent to an iteration.) The initial weights of the network for the program are selected as indicated in this example. The final values of the outputs are equal to -0.0088 and 1.0370. These values are close enough to the desired values. The training error is less than 0.001, which the network has achieved during the training phase.

Example 3.2: Forward Kinematics of Robot Manipulator

In this example a simple back propagation neural network has been used to solve the forward kinematic of a robot manipulator. Therefore, θ_1 θ_2 are the inputs with X, Y as the outputs of the network. A set of 200 samples is applied to the network in the training phase.

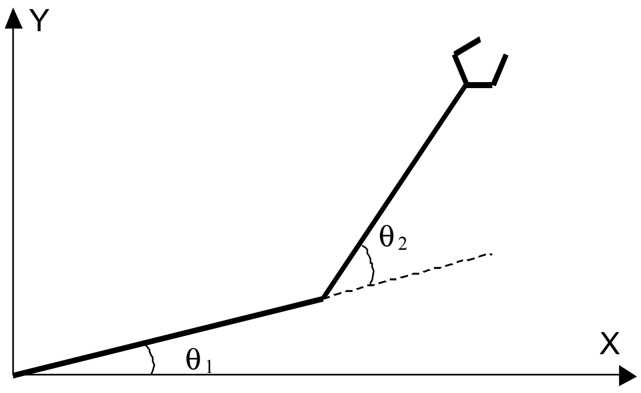


Figure 3.7: The Robot Manipulator.

The relation between (θ_1 and θ_2) and (X and Y) is as follows:

$$\begin{aligned} X &= l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) \\ Y &= l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) \end{aligned} \tag{3.15}$$

Figure 3.8 shows how the error of the network changes until the performance goal has been met.

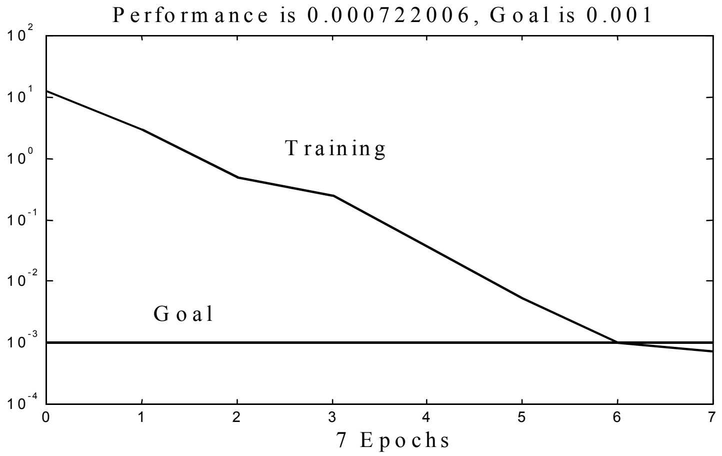


Figure 3.8: The Error of the Network During Training.

After the network has established input and output mapping during the training phase, new inputs are applied to the network to observe its performance in the recall phase. Figure 3.9 shows the simulation result of the network.

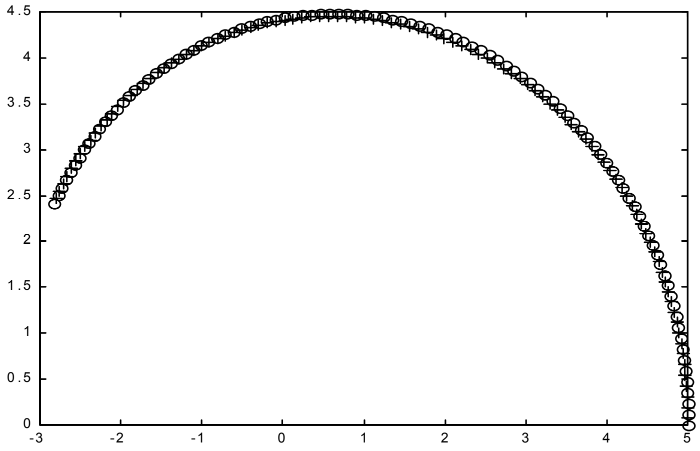


Figure 3.9: The Network Output and Prediction of the Neural Network Using the Back Propagation Algorithm.

3.4 RADIAL BASIS FUNCTION NETWORK (RBFN)

The back propagation method as described in the previous section, has been widely used to solve a number of applications [1],[2]. However, despite the practical success, the back propagation algorithm has serious training problems and suffers from slow convergence [3]. While optimization of learning rate and momentum coefficient parameters yields overall improvements on the networks, it is still inefficient and time consuming for real time applications [4].

Radial Basis Function Networks (RBFN) provide an attractive alternative to BP networks [5]. They perform excellent approximations for curve fitting problems and can be trained easily and quickly. In addition, they exhibit none of the BP's training pathologies such as local minima problems. However, RBFN usually exhibits a slow response in the recall phase due to the large number of neurons associated in the second layer [6],[7]. One of the advantages of RBFN is the fact that linear weights associated with the output layer can be treated separately from the hidden layer neurons. As the hidden layer weights are adjusted through a nonlinear optimization, output layer weights are adjusted through linear optimization.

RBFN approximation accuracy and speed may be further improved with a strategy for selecting appropriate centers and widths of the receptive fields. The redistribution of centers to locations where input training data are meaningful can lead to more efficient RBFN [8].

In this section, the fundamental idea pertaining the RBFN is presented. Furthermore, two examples are provided to clarify the training and recall phases associated with these networks. The network is inspired by Cover's theorem as explained below.

Cover's Theorem[6]: A complex pattern classification problem cast in a high dimensional space nonlinearity is more likely to be linearly separable than in a low dimensional space.

Example 3.3:

Consider the XOR problem as presented previously. As it was shown in chapter 2, an XOR gate cannot be implemented by a single perceptron due to nonlinear separabality property of the input pattern. However, suppose, the following pair of Guassian hidden functions are defined:

$$\begin{aligned} h_1(x) &= e^{-\|x-u_1\|^2} & \qquad u_1 &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ h_2(x) &= e^{-\|x-u_2\|^2} & \qquad u_2 &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned} \tag{3.16}$$

If we calculate $h_1(x), h_2(x)$ for the above input patterns we will have the Table 3.2. Figure 3.10 shows the graph of the outputs in the $h_1 - h_2$ space.

Table 3.2: Mapping of XY to $h_1 - h_2$

Input pattern: X	$h_1(x)$	$h_2(x)$
(1, 1)	1	0.1353
(0, 1)	0.3678	0.3678
(0, 0)	0.1353	1
(1, 0)	0.3678	0.3678

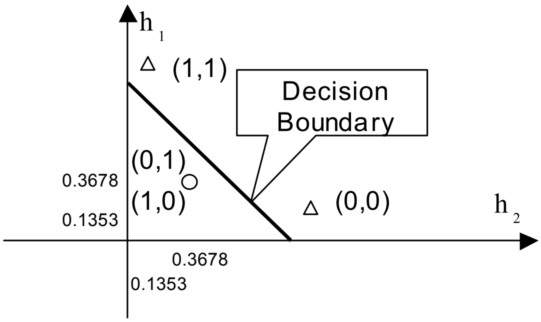


Figure 3.10: XOR Problem in $h_1 - h_2$ Space.

As can be seen, the XOR problem in $h_1 - h_2$ space is mapped to a new problem, which is linearly separable. Therefore, Guassian functions can be used to solve the above interpolation problem with one layer network. The above interpolation problem can be generalized as: Suppose there exist N points $(X_1, \ldots, X_N)$ and a corresponding set of N real values (d_1, d_2, d_3, …, d_1); find a function that satisfies the following interpolation condition:

$$F(x_i) = d_i \qquad i = 1,2,\ldots,N \tag{3.17}$$

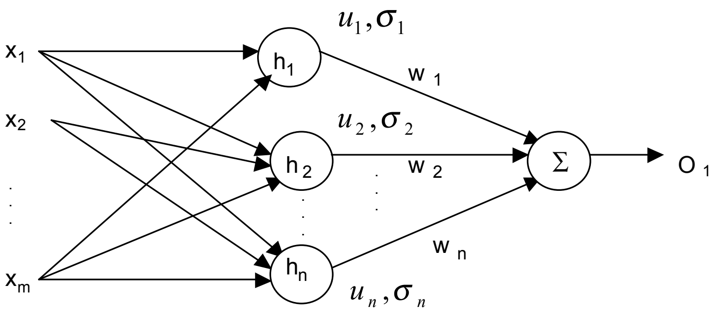


Figure 3.11: A Simple Radial Basis Network.

Figure 3.11 shows a simple radial basis network. This network is a feedforward network similar to back propagation, but it has totally different performance. The first difference is the initial weights. Despite random initial selection of the weights in back propagation, here the initial weights are not chosen randomly. The weights of each hidden layer neuron are set to values that produce a desired response. Such weights are assigned so that the network gives the maximum output for inputs equal to its weights. The activation functions h_i can be defined as follows:

$$h_i = e^{-D_i^2/2\sigma^2} \tag{3.18}$$

where D_i is defined as the distance of the input to the center of the neuron which is identified by the weight vector of hidden layer neuron i. Equation (3.19) shows this relation:

$$\begin{cases} D_i^{\,2} = (x-u_i)^T(x-u_i) \\ x : input \quad vector \\ u_i : Weight \quad vector \quad of \quad hidden \quad layer \quad neuron \quad i \end{cases} \tag{3.19}$$

Therefore, the final contribution of the neuron will decrease for the inputs, which are far from the center of the neuron. With this fact in mind, it is reasonable to give the values of each input of the training set to a neuron, which will result in faster training of the network. The main part of the training of the network is adjusting the weights of the output layer. Figure 3.12 shows a single neuron.

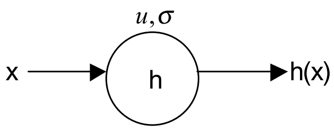


Figure 3.12: A Simple Radial Basis Neuron

Function h(x) as shown in Figure 3.13 can be defined as follows:

$$h(x) = e^{-\frac{(x-u)^2}{2\sigma^2}} \tag{3.20}$$

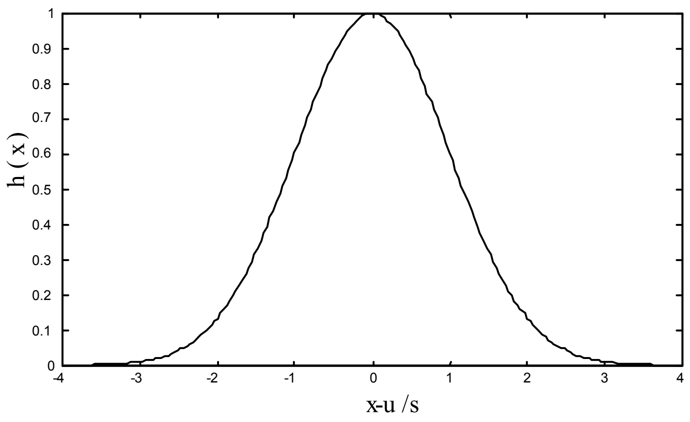


Figure 3.13: The Graph of h(x).

As both graph and formula show:

$$\begin{cases} h(x)=1 & x=u \\ h(x)=0 & |x-u|>3\sigma \\ 0<h(x)<1 & |x-u|<3\sigma \end{cases} \tag{3.21}$$

The above formula indicates that each neuron only possesses contributions from the inputs that are close to the center of the weight function. For other values of x, the neuron will have zero output value with no contribution in the final output of the network. Figure 3.14 shows a radial basis neuron with two inputs, X_1 and X_2.

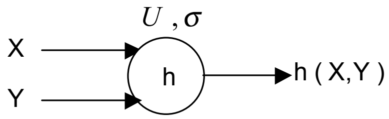


Figure 3.14: A Simple Radial Basis Neuron with Two Inputs.

Figure 3.15 shows the three-dimensional graph of this neuron. As is seen, the fundamental idea is similar. As Figure 3.15 shows, the function is radially symmetric around the center U.

Training of the radial basis network includes two stages. In the first stage, the center U_i and diameter of receptive σ_i of each neuron will be assigned. At the second stage of the training, the weight vector W will be adjusted accordingly. After the training phase is completed, the next step is the recall phase in which the outputs are applied and the actual outputs of the network are produced.

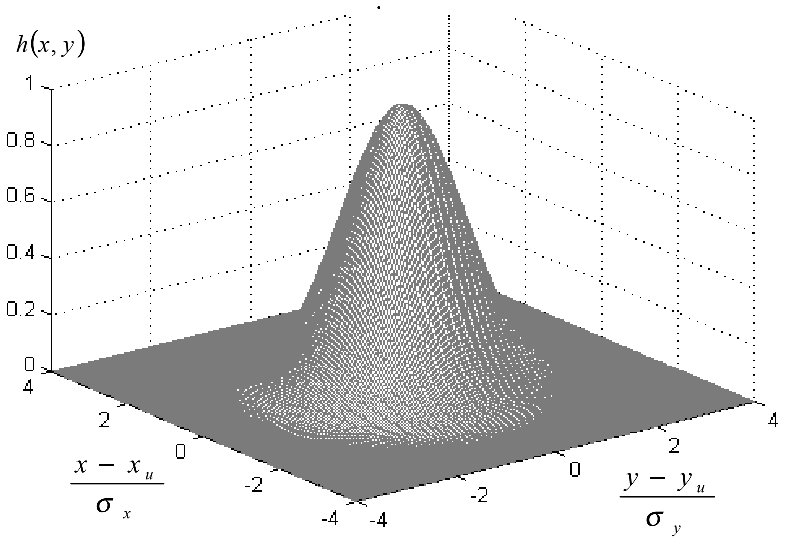


Figure 3.15: Graph of h(x,y) for the Neuron with Two Inputs.

Finding the center U_i of each neuron

One of the most popular approaches to locate the centers U_i is to divide the input vector to some clusters and then find the center of each cluster and locate a hidden layer neuron at that point.

Finding the diameter of the receptive region

The value of σ can have significant effect on the performance of the network. There are different approaches to find this value. One of the popular methods is based on the similarity of the clustering of the input data. For each hidden layer neuron, the RMS distance of each neuron and its first nearest neighbor will be calculated; this value is considered as σ. The training phase of RBFN can be summarized as follows:

1. Apply an input vector X from the training set.
2. Calculate the output of the hidden layer.
3. Compute the output Y and compare it to the desired value. Adjust each weight W accordingly:

$$w_{ij}(n+1) = w_{ij}(n) + \eta.(x_j - y_j)x_i \tag{3.22}$$

4. Repeat steps 1 to 3 for each vector in the training set.
5. Repeat steps 1 to 4 until the error is smaller than a maximum acceptable amount.

The advantage of radial basis network to back propagation network is faster training. The main problem of back propagation is its lengthy training; therefore radial basis networks have caught a lot of attention lately. The major disadvantage of radial basis network is that it is slow in the recall phase due to its nonlinear functions.

Example 3.4:

This example is the same as Example 3.1, where p and o are input and output consecutively. We try to solve the problem using the radial basis network by MATLAB. The details of the program are provided in Chapter 21. The output of the program is shown below. As is observed, the output is very accurate for the same input values. Also, execution of this simple code shows that the network's training is very fast. The answer can be obtained quickly, with high accuracy. The output of the network to a similar input is also shown. $\tilde{o}$ is the output for the new applied input $\tilde{p}$, which is close to p. It can be seen that this value is close to the output of the training input.

$$p = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad o = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \tilde{P} = \begin{bmatrix} 1.1 \\ -0.3 \\ 0.9 \end{bmatrix} \quad \tilde{o} = \begin{bmatrix} 0 \\ 0.9266 \end{bmatrix}$$

Example 3.5:

In this example the inverse kinematics of the robot manipulator of Example 3.2 is solved by RBFN, using MATLAB program. Figure 3.16 compares the actual path and the network prediction of this example. The actual path is shown with circles and the network output with +. As can be seen, the network can predict the path very accurately. In comparison with back propagation, prediction of RBFN is more accurate and the training of this network is much faster. However, due to the number of neurons, the recall phase of the network is usually slower than back propagation.

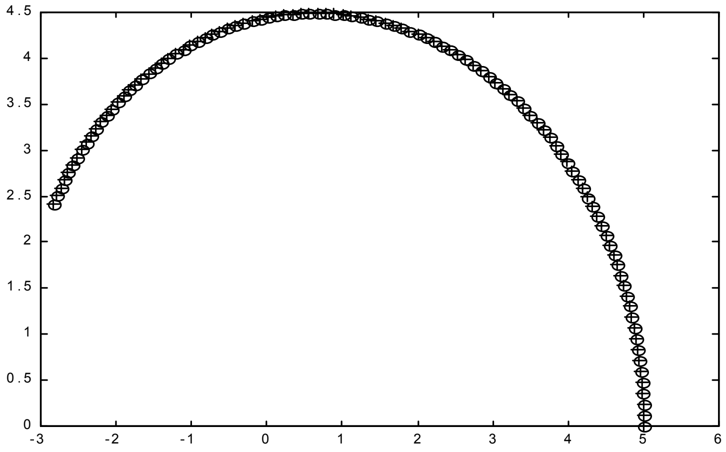


Figure 3.16: Output of the RBFN and Actual Output of the System.

3.5 KOHONEN SELF-ORGANIZATION NETWORK

The Kohonen self-organization network uses unsupervised learning and organizes itself to topological characteristics of the input patterns. The discussion in this section will not seek to explain fully all the intricacies involved in self-organization networks, but rather seek to explain the simple operation of the network with two examples. Interested readers can refer to Kohonen[10], Zurada[11], and Haykin and Simon[12] for more detailed information on unsupervised leaning and self-organization networks.

Learning and brain development phenomena of newborns are very interesting from several viewpoints. As an example, consider how a baby learns to focus its eyes. The skill is not originally present in newborns, but they generally acquire it soon after birth. The parents cannot ask their baby what to do in order to make sense of the visual stimuli impinging on the child's brain. However, it is well known that after a few days, a newborn has learned to associate sets of visual stimuli with objects or shapes. Such remarkable learning occurs naturally with little or no help and intervention from outside. As another example, a baby learns to develop a particular trajectory to move an object or grab a bottle of milk in a special manner. How can these phenomena happen?

One possible answer is provided by a self-learning system, originally proposed by Teuvo Kohonen [10]. His work provides a relatively fast and yet powerful and fascinating model of how neural networks can self-organize. In general, the term *self-organization* refers to the ability of some networks to learn without being given the correct answer for an input pattern. These networks are often closely modeled after neurobiological systems to mimic brain processing and evolution phenomena.

A Kohonen network is not a hierarchical system, but consists of a fully interconnected array of neurons. The output of each neuron is an input to all other inputs in the network including itself. Each neuron has two sets of weights: one set is utilized to calculate the sum of weighted external inputs, and another one to control the interactions between different neurons in the network. The weights on the input pattern are adjustable, while the weights between neurons are fixed.

The other two networks that have been discussed so far in this chapter (BP and RBFN) have neurons that receive input from previous layers and generate output to the next layer or the external world. However, the neurons in the network have neither input nor output to the neurons in the same layer. On the contrary, the Kohonen network receives not only the entire input pattern into the network, but also numerous inputs from the other neurons with the same layer. A block diagram of a simple Kohonen network with N neurons is shown in Figure 3.17.

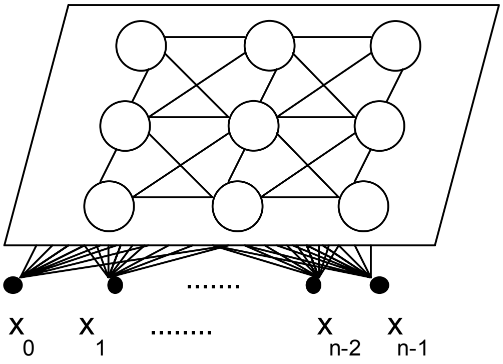


Figure 3.17: A Two Dimensional Kohonen Network.

Notice that the input is connected to all the nodes and there are interconnections between the neurons of the same layer. During each presentation, the complete input pattern is presented to each neuron. Each neuron computes its output as a sigmoidal function on the sum of its weighted inputs. The input pattern is then removed and the neurons interact with each other. The neuron with the largest activation output is declared the winner neuron and only that neuron is allowed to provide the output. However, not only the winning neuron's weight is updated, but also all the weights in a neighborhood around the winning neuron. The neighborhood size decreases slowly with each iteration [11].

3.5.1 Training of the Kohonen Network

When we construct a Kohonen network, we must do two things that have not been generally required by the other networks. First, we must properly initialize the weight vectors of the neurons. Second, the weight vectors and the input vectors should be normalized. These two steps are vital to the success of the Kohonen network. The procedure to train a Kohonen self-organization is as follows:

1. Normalize the random selected weights W_i.
2. Present an input pattern vector x to the network. All neurons in the Kohonen layer receive this input vector.
3. Choose the winning neuron as the one with the largest similarity measure between all weight vectors W_i and the input vector x. If the shortest Euclidean distance is selected as similarity measure within a cluster, then the winning unit m satisfies the following equation:
$$\|x - W_m\| = \min_i \{\|x - w_i\|\} \tag{3.23}$$
where m is referred to as the winning unit.
4. Decrease the radius of Nm region as the training progress, where Nm denotes, as a set of index associated with the winning neighborhood around the winner unit C. The radius of Nm region can be fairly large as the learning starts and slowly reduced to include only the winner and possibly its immediate neighbors.
5. The weight of the winner unit and its neighborhood units are obtained as follows:
$$(W_i)_{new} = (W_i)_{old} + \alpha \lfloor x - (W_i)_{old} \rfloor \tag{3.24}$$
where, Wi is the weight vector, x is the input pattern vector and a is the leaning rate (0<α<1) Since a depend on the size of neighborhood function, Equation (3.25) can be rewritten as
$$(W_i)_{new} = (W_i)_{old} + \alpha\, N_{ci} \lfloor x - (W_i)_{old} \rfloor \tag{3.25}$$
where the function N_c can be chosen appropriately such as a Gaussion function or a Mexican hat function.
6. Present the next input vector. Repeat steps 3−5 until the training phase is completed for all inputs.

In order to achieve a good convergence for the above procedure, the learning rate α, as well as the size of neighborhood Nc should be decreased gradually with each iteration. As was mentioned before, at the beginning of the training phase, the selected region around the winner unit might be fairly large. Therefore, a substantial portion of the network can learn each pattern. As the training proceeds, the size of the neighborhood slowly decreases, so fewer and fewer neurons learn with each iteration. Finally the winner itself will adjust its weights. After the completion of this procedure, the network is trained for the next input vector in a similar fashion.

Kohonen self-organization network has some interesting capabilities that can be extremely useful. One possible application is vector quantization. The network can also be used to perform dimension reduction and feature extraction as well as classification.

In MATLAB, the leaning rate, α, and the neighborhood size are altered through two phases: an ordering phase and a tuning phase.

3.5.2 Examples of Self –Organization

In this subsection, two examples of self-organization maps are provided. The detailed description of examples can be found in chapter 21

Example 3.6: 1-D self-organization Mapping

Consider 200 2-Element unit vectors spread uniformly between 0 and 180 as shown in Figure 3.18. We now consider a 1-D self-organization map with 20 neurons.

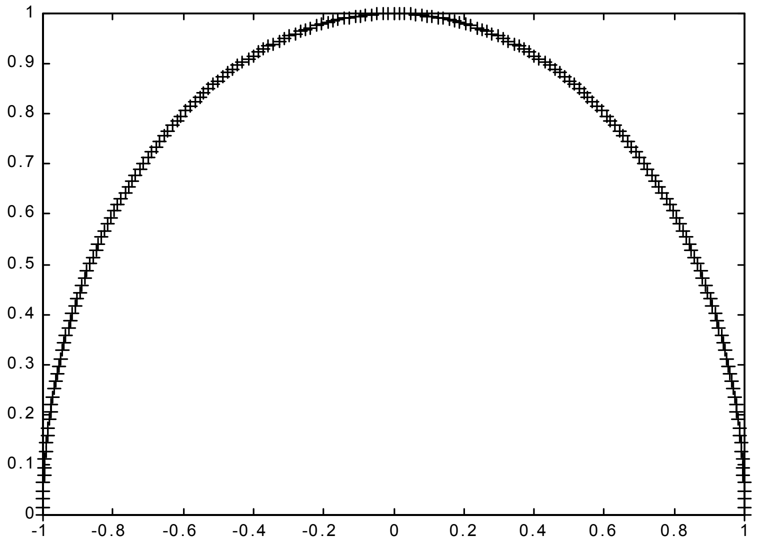


Figure 3.18: Original Distribution of the Input of the Kohonen Network.

Figure 3.19 shows the weights of the Kohonen self-organizing network after training. It can easily be observed that the weights of the network have the pattern of the input. In the other words, the network is being adjusted to the form of the pattern of input of the network.

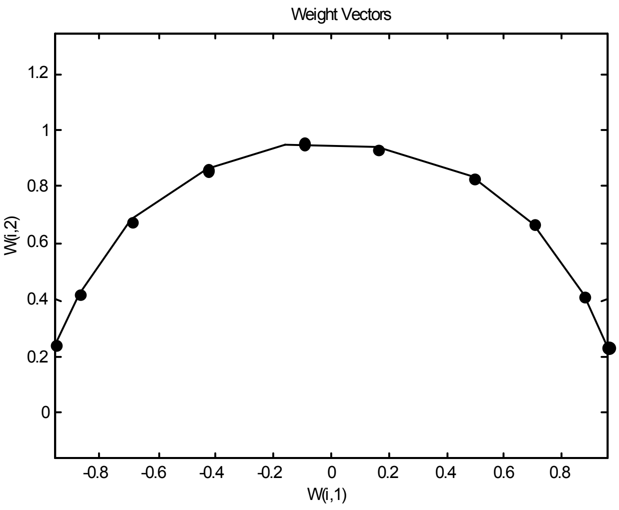


Figure 3.19: Weights of the Kohonen Self-organizing Network.

Example 3.7: 2-D Self-organization Mapping

Suppose we have created 2000 input vectors randomly (Figure 3.20). We will define a two-dimensional map of 35 neurons to classify these input vectors. The two dimensional map is five neurons by seven neurons in horizontal and vertical directions, respectively. The map is then trained for 5,000 presentation cycles in the MATLAB. The results are displayed in Figure 3.22. The details of the program are given in Chapter 21.

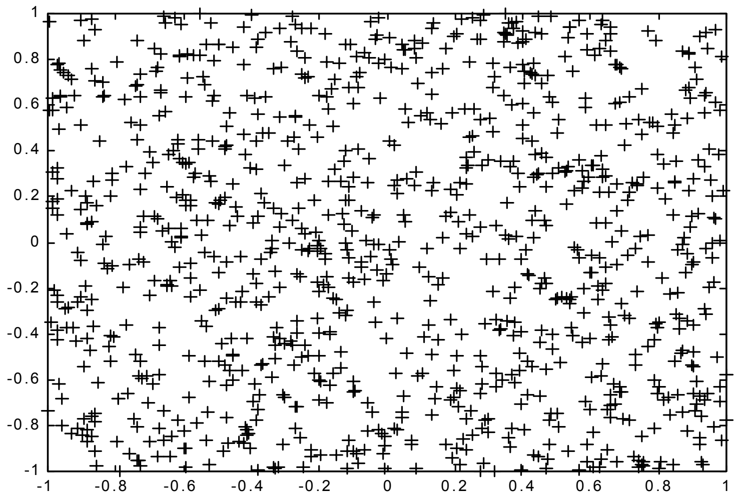


Figure 3.20: Initial Inputs of the Network of Example 3.7.

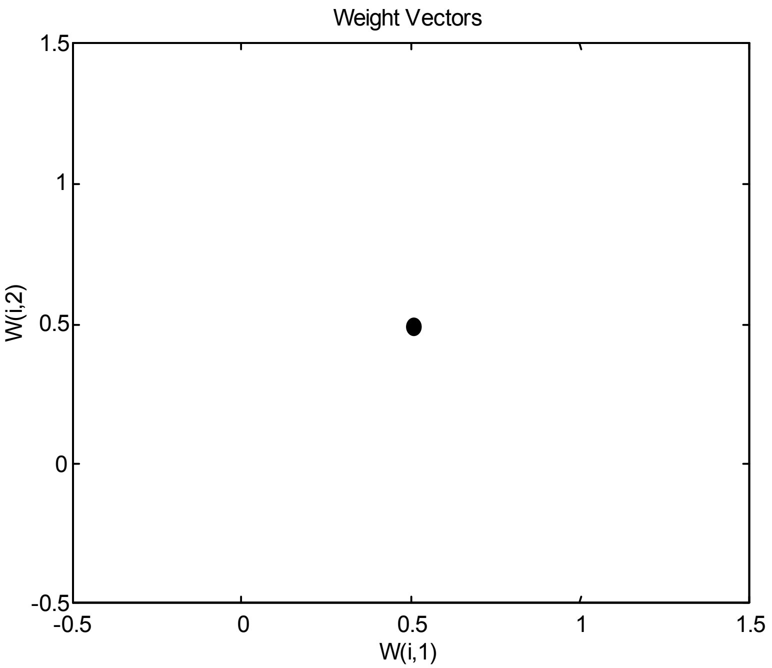


Figure 3.21: Initial Weights of the Network.

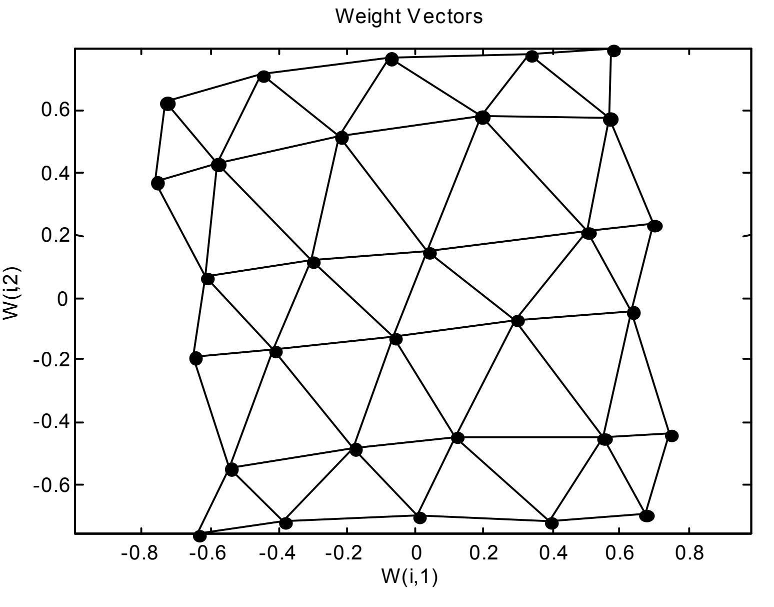


Figure 3.22: Weights of the Kohonen Self-organizing Network after Training (Example 3.7).

3.6 HOPFIELD NETWORK

Hopfield rekindled interest in neural networks by his extensive work on different versions of the Hopfield network [13],[14]. The network can be utilized as an associative memory or to solve optimization problems. One of the original network [13], which can be used as a content addressable memory is described in this chapter. The network is a typical recursive model in which nodes are connected to one another. Figure 3.23 shows a Hopfield network.

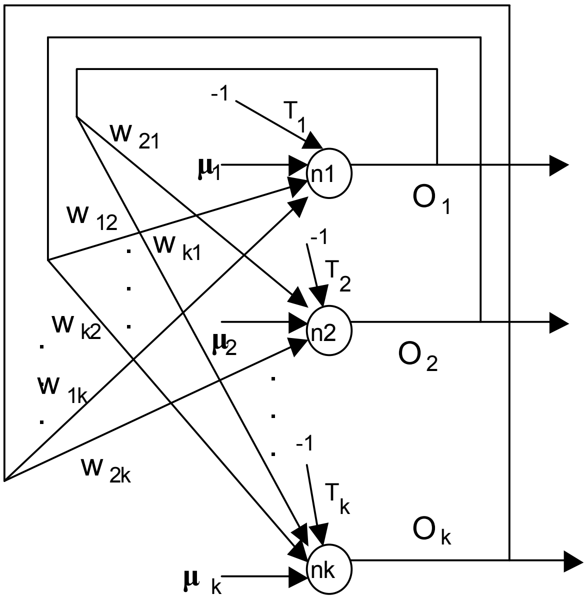


Figure 3.23: Hopfield Network.

As is shown, the output of each neuron consists of the inputs from other neurons, with the exception of itself. Therefore, the activity level of the neurons can be calculated using the following formula:

$$a_i = \sum_{\substack{j=1 \\ j\neq i}}^{n} w_{ij} o_j + \mu_i - T_i \qquad i = 1,\ 2,\ \ldots,\ n \tag{3.26}$$

or in the vector form as:

$$a_i = W_i O + \mu_i - T_i \qquad i = 1,\ 2,\ \ldots,\ n \tag{3.27}$$

where:

$$W_i = \begin{bmatrix} w_{i1} & w_{i2} & \cdots & w_{in} \end{bmatrix} \qquad i = 1,\ 2,\ \ldots,\ n \tag{3.28}$$

Wi is the weight vector for the i-th input of the neural network and the i-th element of this vector is equal to zero. On the other hand,

$$O_i = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_n \end{bmatrix} \qquad i = 1,\ 2,\ \ldots,\ n \tag{3.29}$$

is the output vector of the neural network. Equation 3.27 in the matrix form can be rewritten as follows:

$$A = WO + I - T \qquad i = 1,\ 2,\ \ldots,\ n \tag{3.30}$$

The weight matrix W is a symmetric matrix with all diagonal elements equal to zero. If the activation function of the neuron is a sign function, we will have:

$$o_i = \begin{cases} -1 & if \quad a_i < 0 \\ +1 & if \quad a_i > 0 \end{cases} \tag{3.31}$$

The output transition between old value and new value will happen at certain times. At that time, if the value of the additive weighted sum of a neuron is greater than threshold of that neuron, the new output of that neuron will remain or change to +1, otherwise it will remain or change to –1.

Considering this fact, we can define the state of the network, which is the value of the outputs at one time. For example, $O = [1 \ \ -1 \ \ 1 \ \ \cdots \ \ 1]$ is a state of the network. For each neuron we have two values. Therefore 2^n states exist for a network with n neurons.

In a Hopfield network, we apply an input at certain times and then it will be removed. This causes transitions in states of the network. These transitions continue until the network reaches to a stable point, which is called an *attractor.* An important point about this network is that at each time one neuron will calculate its activity level and change its output. In other words, updating of the outputs of the neuron is being done in an asynchronous fashion. Therefore to calculate activity level of the next neuron, and find the output of that neuron, we use some updated value for the output of the other neurons. The updating order of the neurons is random. It depends on random propagation delays and noise. When using the formula in matrix form, we should be careful, because it offers synchronous or parallel updating. If we consider $E = [1 \ \ -1]$, each state of the system is an edge of the graph in E^n space. After applying an input pattern, the state of the network goes from edge to adjacent edge until it reaches an attractor of 2^n edges. An attractor should satisfy the equation:

$$\mathrm{sgn}[A_a] = O_a \tag{3.32}$$

Where A_a and O_a are activity level and output at the attractor. Note that if the network satisfies this equation, the next state of the network is equal to its present state and therefore no transition will happen until a new input pattern is applied to the network.

As mentioned earlier, input will be applied momentarily and then will be removed. Considering this fact and using Equation 3.30, Equation 3.32 will change to:

$$O_a = \operatorname{sgn}[WO_a - T] \tag{3.33}$$

If we define the energy function for the system as:

$$E = -\frac{1}{2}O^T WO + \mu \cdot O - T^T O = -\frac{1}{2}\sum_{\substack{i=1 \\ j\neq i}}^{n}\sum_{j=1}^{n} w_{ij}o_i o_j - \sum_{i=1}^{n} i_i o_i + \sum_{i=1}^{n} T_i o_i \tag{3.34}$$

The gradient of the energy can be calculated from Equation 3.34 as:

$$\nabla E = -\frac{1}{2}(W^t + W)O - \mu^T + T^T = -WO - \mu^T + T^T \tag{3.35}$$

Here we have used the fact that the weight matrix is symmetric. The energy increment is equal to:

$$\Delta E = (\nabla E)^T \Delta O \tag{3.36}$$

As discussed earlier outputs will be updated one at a time. Therefore only i-th output will be updated,

$$\Delta O = \begin{bmatrix} 0 & \cdots & o_i & \cdots & 0 \end{bmatrix}^T \tag{3.37}$$

The energy increment will be equal to:

$$\Delta E = (-W_i^T O - \mu_i^T + T_i^T)\Delta o_i = -A_i \Delta o_i \tag{3.38}$$

It is obvious that for positive A_i, $\Delta o_i \geq 0$ and for negative A_i, $\Delta o_i \leq 0$. Looking at Equation 3.38 it can be seen that $\Delta E \leq 0$. Therefore it can be concluded that state transitions of the network are in a way that the energy is either decreased or retained. This means that the attractors are the edges with lowest levels of energy. Following is an example to clarify these ideas.

Example 3.8:

Shows the state transitions and attractors in a fourth order Hopfield network. Consider the weight matrix as follows:

$$W = \begin{bmatrix} 0 & -1 & -1 & 2 \\ -1 & 0 & 1 & -1 \\ -1 & 1 & 0 & -1 \\ 2 & -1 & -1 & 0 \end{bmatrix} \tag{3.39}$$

Considering the threshold and external inputs equal to zero, energy level can be calculated as follows:

$$E = \frac{1}{2}O^T WO \tag{3.40}$$

or:

$$E = -\frac{1}{2}\begin{bmatrix} o_1 & o_2 & o_3 & o_4 \end{bmatrix}\begin{bmatrix} 0 & -1 & -1 & 2 \\ -1 & 0 & 1 & -1 \\ -1 & 1 & 0 & -1 \\ 2 & -1 & -1 & 0 \end{bmatrix}\begin{bmatrix} o_1 \\ o_2 \\ o_3 \\ o_4 \end{bmatrix} \tag{3.41}$$

After simplification we will have:

$$E = -o_1\left(-o_2 - o_3 + 2o_4\right) - o_2\left(o_3 - o_4\right) + o_3 o_4 \qquad (3.42)$$

Now if we consider all the states of the network starting from $\begin{bmatrix}-1 & -1 & -1 & -1\end{bmatrix}$ to $\begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix}$, we can calculate all the energy levels of the network. The result will be the levels 1, 1, -1 3, -1, 3, -7, 1, 1, -7, -1, 3, 3, -1, 3, 1 respectively. Therefore the energy levels are –7, -1, 1, 3. The two states with the lowest energy level -7 are $\begin{bmatrix}-1 & 1 & 1 & -1\end{bmatrix}$ and $\begin{bmatrix}1 & -1 & -1 & 1\end{bmatrix}$. We can see that these states are attractors of the network. In other words, they satisfy Equation 3.33. If we try any other state of the network, we will see that they do not satisfy this equation, which means that they are not attractors of the network.

In other words, the attractors are the states with minimum levels of energy. In fact we can see that the transition in the network will be from an state to another state with a lower or the same level of energy. On the other hand, we know that the transition is asynchronous. Therefore, at each single step we will go from one state to its adjacent state. These transitions are in the direction of reduction of energy level until we reach a state with a minimum level of energy, which is the attractor of the network. Figure 3.24 shows the state transition of the network.

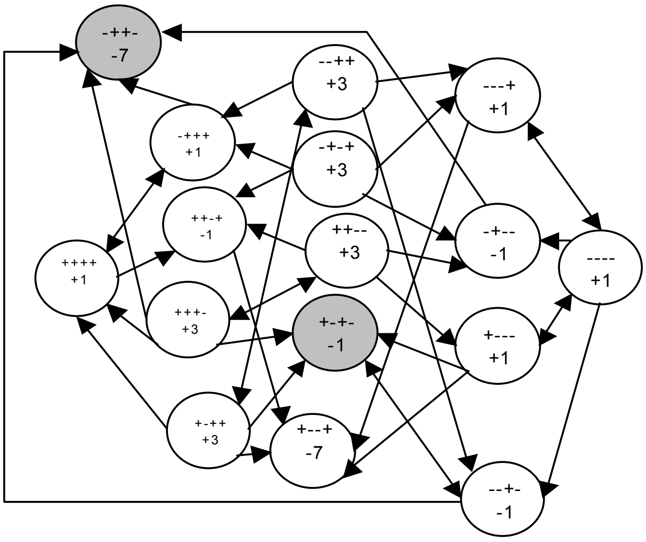


Figure 3.24: State Transition of the Hopfield Network to Reach to a Stable State.

3.7 CONCLUSIONS

In this chapter, four different neural networks were presented. Several numerical examples were provided to demonstrate the effectiveness of these networks. The described networks consist of highly parallel building blocks that illustrate NN design principles. They can be used to construct more complex systems. In general, the NN architectures cannot compete with the conventional techniques at performing precise numerical operations. However, there are large classes of problems that often involve ambiguity, prediction, or classifications that are more amenable to solution by NN than other available techniques. In the following chapters several of these problems will be addressed in detail.

REFERENCES:

1. Widrow, B. and Lehr, M. A., Thirty Years of Adaptive Neural Networks: Perceptron, MADALINE, and back propagation. *Proc. of the IEEE,* Vol. 78, 1415-1442, 1990.
2. Rumelhart, D.E., Hinton G.E. and Williarns, R.J., Learning Internal Representations by Error Propagation, *Parallel Data Processing,* Vol. 1, Chap. 8, the MIT Press, Cambridge, MA, 1986.
3. Specht, D.F., A General Regression Neural Network*, IEEE Trans. on Neural Networks,* Vol. 2, 568–576, 1991.
4. Wasserman P.D., *Advanced Methods in Neural Computing*, Van Nostrand Reinhold, New York, 1993.
5. Moody, J. and Darken, C., Fast Learning in Networks of Locally-Tuned Processing Units, *Neural Computation*, Vol. 1, 281–294, 1989.
6. Jang, J. S., Sun, C. T., and Mizutani, E., *Neuro-Fuzzy and Soft Computing*, Prentice Hall, Englewood Cliffs, NJ, 1997.
7. Lowe, D., Adaptive Radial Basis Function Nonlinearities and the Problem of Generalization, *Proc. First IEEE Int. Conf. on Artificial Networks*, London, UK, 1989.
8. Wettschereck D. and Dietterich, T., Improving the Performance of Radial Basis Function Networks by Learning Center Locations, *Advances in Neural Information Processing Systems*, Vol. 4, 1133-1140, Morgan Kaufmann, San Mateo, CA, 1992.
9. Nguyen, D. and Widrow B., The Truck Backer-Upper, *Int. Joint Conf. on Neural Networks*, Washington, DC, Vol. 2, 357–363, 1989.
10. Kohonen, T., *Self-organization and Associative Memory*, 3rd ed., Springer-Verlag, New York, 1988.
11. Zurada, J., *Introduction to Artificial Neural Systems*, West Publishing Co, St. Paul, MN, 1992.
12. Haykin, S., *Neural Networks: A Comprehensive Foundation*, Prentice Hall, Upper Saddle River, NJ, 1999.
13. Hopfield J.J., Neurons with Grades Response Have Collective Computational Properties Like Those of Two State Neurons", *Proc.* of *National* Academic *of Science*, Vol. 81, 3088–3092, 1984.
14. Hopfield, J.J. and Tank, D.W., Computing with Neural Circuits: A Model, *Science*, Vol. 233, 625–633, 1986.